AUTOMATIC CACHING GENERATION IN NETWORK APPLICATIONS

FIELD OF THE INVENTION

[0001] Embodiments of the invention relate to network applications; and more specifically, to automatic cache generation for network applications.

BACKGROUND OF THE INVENTION

[0002] Network processors (NP) are emerging as a core element of high-speed communication routers and they are designed specifically for packet processing applications. Such applications usually have stringent performance requirements. For instance, OC-192 (10 Gigabits/sec) POS (Packet over SONET) packet processing requires a throughput of 28 million packets per second or service time of 4.57 microseconds per packet for transmission and receipt in the worst case.

[0003] On the other hand, the latency for an external memory access in NPs is usually larger than the worst-case service time. In order to address the unique challenge of packet processing, (e.g., maintaining stability while maximizing throughput and minimizing latency for the worst-case traffic,) modern network processors usually have a highly parallel architecture. For instance, some network processors, such as, Intel IXA NPU family of network processors (IXP), includes multiple microengines (e.g., programmable processors with packet processing capability) running in parallel and each microengine supports multiple hardware threads.

[0004] Consequently, the associated network applications are also highly parallel and usually multi-threaded to compensate the long memory access latency. Whenever a new packet arrives, a series of tasks (e.g., receipt of the packet, routing table look-up, and enqueueing) is performed on that packet by a new thread. In such a parallel programming

paradigm, modifications to global resources, such as a location in the shared memory, are protected by critical sections to ensure mutual exclusiveness and synchronization between threads.

[0005] Each critical section typically reads a resource, modifies it, and writes it back (RMW). Figure 1 is a block diagram illustrating a conventional external memory accesses by multiple threads. As shown in Figure 1, if more that one thread is required to modify the same critical data, a latency penalty will be incurred for each thread if each accesses the external memory. Referring to Figure 1, each of the threads 101-104 has to be executed in sequence. For example, thread 102 has to wait thread 101 to finish the operations read, modification, and write back to the external memory before thread 102 can access the same location of the external memory.

BRIEF DESCRIPTION OF THE DRAWINGS

[0006] The invention may best be understood by referring to the following description and accompanying drawings that are used to illustrate embodiments of the invention. In the drawings:

[0007] Figure 1 is block diagram illustrating a typical external memory access.

[0008] Figure 2 is a block diagram illustrating an example of an external memory access using software controlled caching according to one embodiment.

[0009] Figure 3 is a block diagram illustrating an example of a caching mechanism according to one embodiment.

[0010] Figure 4 is a block diagram illustrating an example of a caching mechanism according to an alternative embodiment.

[0011] Figure 5 is a flow diagram illustrating an example of a process for software automatic controlled caching, according to one embodiment.

[0012] Figures 6-8 are block diagrams of examples of pseudo code illustrating software controlled caching operations according to one embodiment.

[0013] Figure 9 is a flow diagram illustrating an example of a process to identify a candidate for caching according to one embodiment.

[0014] Figures 10-12 are block diagrams of examples of pseudo code illustrating software controlled caching operations according to one embodiment.

[0015] Figure 13 is a block diagram illustrating an example of an external memory access using software controlled caching according to one embodiment.

[0016] Figure 14 is a block diagram illustrating an example of memory allocations of CAM and LM according to one embodiment.

[0017] Figure 15 is a flow diagram illustrating an example of a process for automatic software controlled caching according to one embodiment.

[0018] Figure 16 is a flow diagram illustrating an example of a process for maintaining images of a CAM and/or LM within a microengine, according to one embodiment.

[0019] Figure 17 is a block diagram of an example of pseudo code for the process example of Figure 16.

[0020] Figure 18 is a block diagram of an example of a processor having multiple microengines according to one embodiment.

[0021] Figure 19 is a block diagram illustrating an example of a data processing system according to one embodiment.

DETAILED DESCRIPTION

[0022] Automatic software controlled caching generations in network applications are described herein. In the following description, numerous specific details are set forth. However, it is understood that embodiments of the invention may be practiced without these specific details. In other instances, well-known circuits, structures and techniques have not been shown in detail in order not to obscure the understanding of this description.

[0023] Some portions of the detailed descriptions which follow are presented in terms of algorithms and symbolic representations of operations on data bits within a computer memory. These algorithmic descriptions and representations are used by those skilled in the

data processing arts to most effectively convey the substance of their work to others skilled in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of operations leading to a desired result. The operations are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical or magnetic signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0024] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the following discussion, it is appreciated that throughout the description, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar data processing device, that manipulates and transforms data represented as physical (e.g. electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers or other such information storage, transmission or display devices.

[0025] Embodiments of the present invention also relate to apparatuses for performing the operations described herein. An apparatus may be specially constructed for the required purposes, or it may comprise a general purpose computer selectively activated or reconfigured by a computer program stored in the computer. Such a computer program may be stored in a computer readable storage medium, such as, but is not limited to, any type of disk including floppy disks, optical disks, CD-ROMs, and magnetic-optical disks, read-only memories (ROMs), random access memories (RAMs) such as Dynamic RAM (DRAM), erasable programmable ROMs (EPROMs), electrically erasable programmable ROMs (EEPROMs), magnetic or optical cards, or any type of media suitable for storing electronic instructions, and each of the above storage components is coupled to a computer system bus.

[0026] The algorithms and displays presented herein are not inherently related to any particular computer or other apparatus. Various general purpose systems may be used with programs in accordance with the teachings herein, or it may prove convenient to construct more specialized apparatus to perform the methods. The structure for a variety of these systems will appear from the description below. In addition, embodiments of the present invention are not described with reference to any particular programming language. It will be appreciated that a variety of programming languages may be used to implement the teachings of the embodiments of the invention as described herein.

[0027] A machine-readable medium includes any mechanism for storing or transmitting information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes read only memory ("ROM"); random access memory ("RAM"); magnetic disk storage media; optical storage media; flash memory devices; electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, etc.); etc.

[0028] In one embodiment, Software controlled caching can be used to help reduce the latency penalty, by folding the multiple RMW operations into a single read and one or more modifications. Consequently, the number of external memory accesses is significantly reduced, and the latency caused by the inter-thread dependence (around the critical data) is effectively minimized.

[0029] Figure 2 is a block diagram illustrating an example of external memory accesses by multiple threads, according to one embodiment. The external memory access example 200 includes multiple threads including threads 201-204. Similar to memory accesses shown in Figure 1, each thread includes a read operation, a modification operation, and a write operation, also referred to as RMW operations. In one embodiment, the RMW operations of threads 201-204 are merged or folded into a single read operation, one or more modification operations, a single write operation at the end, using a caching mechanism, which will be described in details further below. As a result, the number of external memory accesses is significantly reduced, and the latency caused by the inter-thread dependence is greatly reduced.

[0030] Figure 3 is a block diagram illustrating an example of a caching mechanism according to one embodiment. In one embodiment, the caching mechanism 300 may be implemented within a microengine of a processor having multiple microengines, such as, for example, Intel IXA NPU family of network processors (IXP).

[0031] For example, according to one embodiment, in each microengine of a processor having multiple microengines, such as, Intel IXA NPU family of network processors (IXP), a content addressable memory (CAM) unit and local memory (LM) may be combined to implement the software controlled caching. The CAM unit in a microengine is the content addressable memory. Each of the entries in the CAM unit stores the state and tag portion of a cache line and its least recently used (LRU) logic maintains a time-ordered list of CAM entry usage. In addition, the local memory in a microengine is basically an indexed register file, in which the data portion of cache lines can be stored. All the related cache operations (e.g., lookup for the tag, load of the data, write back of the data, etc.) are under software control.

[0032] In one embodiment, a processor includes, but not limited to, multiple microengines having a content addressable memory (CAM) and a local memory respectively to perform multiple threads substantially concurrently, each of the threads including one or more instructions performing at least one external memory access based on a base address that is substantially identical, where the base address is examined in the CAM to determine whether the CAM includes an entry containing the base address, and an entry of the local memory corresponding to the entry of the CAM is accessed without having to accessing the external memory, if the CAM includes the entry containing the base address.

[0033] Referring to Figure 3, in one embodiment, the caching mechanism example 300 includes, but not limited to, a CAM 301 and a LM 302. Although, the caching mechanism example 300 illustrates one CAM and one LM, it will be appreciated that more than one CAM and LM may be implemented. The Cam 301 and LM 302 may be used, in combination, to automatically implement software controlled caching to reduce the latency penalty of accessing external memory in network applications. For the purposes of illustrations, the CAM and LM units are used as an example to illustrate the program transformation

throughout the present application. However, it will be appreciated that other types of memory and mechanisms may also be implemented without departing broader scope and spirit of embodiments of the invention.

[0034] In one embodiment, CAM 301 includes one or more entries 304-306, where each of the entries 304-306 includes a tag field and a state field to store the tag portion and the state portion of a cache line respectively. In addition, CAM 301 further includes a least recently used (LRU) logic to determine the least recently used entry of CAM 301. In one embodiment, LM 302 includes one or more entries 308-310, where each of t he entries 308-310 is used to store a data portion of a cache line.

[0035] In one embodiment, when a request for accessing location having a base address of an external memory is received, the microengine that handles the thread of the request may examine (e.g., walking through) the CAM 301 to locate an entry having the requested base address. For example, each of the entries 304-306 of CAM 301 may be examined to match the requested base address of the external memory. Typically, the entries 304-306 of CAM 301 may store the base addresses of recently accessed external memory. If the entry having the requested base address is found in CAM 301, LRU logic 303 returns a result 311 having state field 312, status field 313, and entry number field 314. The state field 312 may contains the state of the corresponding cache line and/or the state of CAM, and status field 313 indicating whether the cache is a hit (result 311b) or a miss (result 311a). The entry number field 314 contains the hit entry number of the CAM 301 that contains the requested base address. The hit entry number may be used to access an entry of LM 302 according to a predetermined algorithm, which will be described in details further below, via index logic 307 of LM 302.

[0036] If it is determined that the CAM 301 does not contains the requested base address, LRU logic 303 returns a least recently used entry of CAM 301 (e.g., result 311a). The least recently used entry of the CAM is linked to an entry of LM 302 that may be used to store (e.g., caching) the data from the external memory access for subsequent external memory accesses to the same base address.

[0037] The LRU logic 303 may be implemented as a part of CAM 301 and/or index logic may be implemented as a part of LM 302. However, the configurations may not be limited to the one shown in Figure 3. It will be appreciated that other configurations may exist. For example, according to an alternative embodiment, the LRU logic 303 may be implemented external to CAM 301 and/or the index logic 307 may be implemented external to LM 302, as caching mechanism example 400 shown in Figure 4. Further, the CAM 301 and/or LM 302 may be implemented as a part of a global CAM and/or LM that are shared among multiple microengines of a processor, where the global CAM and/or LM may be partitioned into multiple partitions for multiple microengines. Other configurations may exist.

[0038] Figure 5 is a flow diagram illustrating an example of a process for software automatic controlled caching, according to one embodiment. Exemplary process 500 may be performed by a processing logic that may comprise hardware (circuitry, dedicated logic, etc.), software (such as is run on a dedicated machine), or a combination of both. For example, process example 500 may be performed by a compiler when compiling source code written in a variety of programming languages, such as, for example, C/C++ and/or assembly. In one embodiment, process example includes, but not limited to, identifying a candidate representing a plurality of instructions of a plurality of threads that perform one or more external memory accesses, the external memory accesses having a substantially identical base address, and inserting one or more directives or instructions into an instruction stream corresponding to the identified candidate to maintain contents of at least one of a content addressable memory (CAM) and local memory (LM) of a processor and to modify at least one of the external memory access to access at least one of the CAM and LM of the processor without having to perform the respective external memory access.

[0039] Referring to Figure 5, at block 501, processing logic receives source code written in a variety of programming languages, such as, for example, C/C++ and/or assembly. At block 502, the processing logic parses the source code to identify one or more candidates for software controlled cache, such as, for example, external memory accesses in each thread having substantially identical base address. At block 503, the processing logic inserts one or more instructions into the instructions steam of the source code to maintain images of a

CAM and/or LM of a microengine of a processor, and to modify the original external memory access of the candidates to access the data images of the LM without having to access the external memory. Other operations may also be performed.

[0040] In one embodiment, after the network application is multi-threaded, either manually or automatically through a parallelizing compiler, each thread performs essentially the same operations on a newly received packet, and modifications to global resources (e.g., external memory accesses) are protected by critical sections. This transformation automatically recognizes the candidate (external memory accesses in each thread) for caching, and implements the software controlled caching (e.g., maintaining the CAM and LM images, and modifying the original accesses to access the data image in LM).

[0041] In order to provide a candidate for software controlled caching, the candidate has to be identified out of multiple potential candidates based on an analysis of the corresponding source code. Figure 6 is an example of a source code, which may be used to provide candidates for software controlled caching according to one embodiment. Referring to Figure 6, source code example 600 includes multiple operation blocks 601-604 that may be processed via multi-threading processes.

[0042] In one embodiment, source code example 600 may be analyzed to generate a pool of potential candidates, as shown in Figure 7. Referring to Figure 7, in this example, the pool of potential candidates 700 include candidates 701-705. In one embodiment, candidates 701-705 may be analyzed to identify a closed set of accesses the external memory via the same base address. That is, they access addresses (base + offset), where base is common to all the accesses in the set and may be different for different threads, and offset is a constant. This set is closed in the sense that it contains all accesses in the thread that are in a dependence relation with each other.

[0043] As a result, as shown in Figure 8 according to one embodiment, any ineligible candidates may be screened out and one or more eligible candidates may be merged into a larger candidate based on their identical base address. In one embodiment, a candidate is ineligible for caching if the base is not common to all the accesses in the candidate. Alternatively, if the memory locations accessed by a candidate can be accessed by other

programs, the respective candidate is ineligible. In this example, candidate 701 is an ineligible candidate since candidate 701 has a different base address. On the other hand, candidates 702-705 are eligible candidates because they have the identical base address (e.g., state + 16*i) and thus, candidates 702-705 are merged into a single larger candidate 800 as a final candidate for caching.

[0044] Figure 9 is a flow diagram illustrating an example of a process to identify a candidate for caching according to one embodiment. The process example 900 may be performed by a processing logic that may comprise hardware (circuitry, dedicated logic, etc.), software (such as is run on a dedicated machine), or a combination of both. For example, process example 900 may be performed by a compiler when compiling source code written in a variety of programming languages, such as, for example, C/C++ and/or assembly.

[0045] Referring to Figure 9, at block 901, the processing logic partitions substantially all of the external memory accesses into one or more set of candidates (e.g., closed sets). For example, if access A depends on access B, they may be grouped in the same partition. That is, if two accesses are in a dependence relation, they should access the same data image (either in the external memory or in the local memory). Hence each closed set identified is a candidate for caching.

[0046] At block 902, one or more copy-forward transformations may be optionally performed on the addresses of each external memory access. For example, the following operations:

```
a = b + c;
d = load [a];
```

may be transformed into the following operation:

```
d = load [b + c]
```

[0047] At block 903, one or more global value numbering and/or constant folding operations may be performed for each thread. For example, during a global value numbering operation, the following operations:

```
a = 2;
b = c*a;
```

```
d = 2;
e = d*c;
```

may be transformed into the following operations:

```
a = 2;
b = c*a;
a = 2;
b = c*a;
```

[0048] For example, during a constant folding operation, the following operations:

```
a = 2;
b = c + d;
e = a + b;
```

may be transformed into the following operations:

```
a = 2;
b = c + d;
e = 2 + b;
```

[0049] At block 904, for each external memory access of each candidate, the address of the external memory access is converted into a form of (base + offset). In one embodiment, the base address is a non-constant part and the offset is a constant part of the address. Note that if a program has been value numbered in the sense that identical addresses are made to have identical representations, the effectiveness of the transformation will be improved.

[0050] At block 905, one or more eligible candidates, such as, for example, candidates 702-705 of Figure 7, are identified, while the ineligible candidates are screened out (e.g., eliminated). In one embodiment, if the base address is not identical over substantially all the external memory accesses in the partition, the corresponding candidate or candidates are considered as ineligible. Further, if the memory locations of the external memory accesses may be accessed by other programs, the corresponding candidate or candidates may be considered as ineligible. That is, if the memory addresses being accessed are known to other programs, such as, for example, externally defined, they are not suitable for caching. Such candidate or candidates may be identified via an escape analysis performed by a compiler.

[0051] At block 906, the eligible candidates may be consolidated into a single large candidate. For example, according to one embodiment, all of the eligible candidates having the identical base address may be grouped into a single candidate, such as candidate 800 of Figure 8. At block 907, a candidate having largest candidates grouped is selected as a final candidate for caching. Other operations may also be performed.

[0052] Note that the software controlled caching is used to reduce the latency penalty of modifications to global resources by multiple threads. However, if the CAM and LM units are global resources shared by multiple threads, all the caching operations should be protected by a critical section. According to one embodiment, in order to simplify the synchronization needed for the critical section, only one candidate (e.g., one global resource) is selected. However, it will be appreciated that more than one candidate may be selected dependent upon a particular system design, as long as the critical section is handled appropriately.

[0053] After the final candidate for caching is identified, one or more instructions are inserted into the corresponding instruction stream to maintain the CAM and LM units, and to modify the original accesses in the candidate to access the data image in LM. Figure 10 is a block diagram illustrating an example of pseudo code that a caching instruction has been inserted before each external memory access instruction. In some cases, the inserted caching instructions may be duplicated and can be consolidated into one caching instruction per related external memory accesses, as shown in Figure 11. Thereafter, the inserted caching instructions may be expanded to modify the external memory accesses to access the corresponding local memory instead of accessing external memory, as shown in Figure 12. In this example, base = state + i * 16, m = 0, M = 15 and n is the entry in CAM that contains the base.

[0054] Consequently, when different threads access the same data, they can directly access the data image in LM, as illustrated in Figure 13 (assume the starting address B in LM is 0). In addition, some further optimizations may be performed. For example, in response to a CAM lookup miss, the expanded caching instructions could only write back the dirty bytes and load only the bytes actually used. Other operations may also be performed.

[0055] In order to perform appropriate software controlled caching, sufficient local memory space has to be reserved for each external memory access. Figure 14 is a block diagram illustrating an example of memory allocations of CAM and LM according to one embodiment. Referring to Figure 14, memory configuration example 1400 includes a CAM 1401 and a local memory 1402. CAM 1401 includes one or more entries 1403-1404 which correspond to one or more memory spaces 1405-1406 respectively reserved.

[0056] For each the access in the candidate, its address is in the form of base + offset. For the purposes of illustrations, if m is the minimum address byte accessed and M is the maximum address byte accessed over all the accesses in the selected candidate (having n^{th} entry of CAM 1401), and N is the number of entries in CAM, then N*(M-m+1) bytes may be reserved in LM 302 for the data image. Assume the stating address of the data images in LM 302 is B. If the base address of the candidate is stored in the n^{th} entry 1404 in the CAM, where n is ranging from 0 to N-1, then the data portion of the associated cache line in LM 302 is from B + n*(M-m+1) to B + (n+1)*(M-m+1) – 1, as indicated in memory space 1406.

[0057] Figure 15 is a flow diagram illustrating an example of a process for automatic software controlled caching according to one embodiment. Process example 1500 may be performed by a processing logic that may comprise hardware (circuitry, dedicated logic, etc.), software (such as is run on a dedicated machine), or a combination of both. For example, process example 1500 may be performed by a compiler when compiling source code written in a variety of programming languages, such as, for example, C/C++ and/or assembly.

[0058] Referring to Figure 15, at block 1501, a sufficient memory space is reserved in a local memory to store the data portions of the cache lines of the external memory accesses. At block 1502, a caching instruction is inserted before each of the external memory access instruction for each candidate, as shown, for example, in Figure 10. At block 1503, the processing logic optionally performs partial redundancy elimination operations over the inserted caching instructions, such that after the partial redundancy elimination, there is at most one caching instruction in any path through the program, as shown, for example, in Figure 11. At block 1504, the caching instructions are expanded into one or more code sequences for looking up the base address in the CAM, evicting the old cache lines, and

loading the new image to the LM upon a lookup miss in CAM, etc. At block 1505, each of the external memory accesses of each candidate is modified to access the associated data image in the LM. Note that all the instructions implementing the software controlled caching, including the expanded caching instructions and the modified accesses, should be protected by a critical section. Other operations may also be performed.

[0059] Figure 16 is a flow diagram illustrating an example of a process for maintaining images of a CAM and/or LM within a microengine, according to one embodiment. Process example 1600 may be performed by a processing logic that may comprise hardware (circuitry, dedicated logic, etc.), software (such as is run on a dedicated machine), or a combination of both. For example, process example 1600 may be performed by a compiler when compiling source code written in a variety of programming languages, such as, for example, C/C++ and/or assembly. The process example 1600 may be performed as a part of operations performed at block 1504 of Figure 15.

[0060] Referring to Figure 16, when the processing logic receives a request to access an external memory location, at block 1601, the processing logic looks up in the CAM to determine whether the CAM contains a base address of the requested external memory access. If the CAM contains the base address of the requested external memory access (e.g., a hit), at block 1606, an index of the corresponding entry of the CAM that contains the base address is retrieved, and at block 1607, an entry of the LM is accessed based on the retrieved index without having to access the external memory.

[0061] If the CAM does not contains the base address of the requested external memory access (e.g., a miss), at block 1602, a least recently used (LRU) entry of the CAM is allocated or identified (e.g., an entry for previous caching operation of a previous external memory access), and the index and the address stored therein are retrieved. At block 1603, the retrieved address stored in the LRU entry of the CAM is examined to determine whether the address is valid.

[0062] If the address is determined to be valid, at block 1604, the data stored in the corresponding local memory (e.g., the previous cached data for a previous external memory access) is written back (e.g., swapped) into the external memory based on the identified valid

address. Thus, the LRU entry of the CAM and the corresponding LM space are now available for caching the current external memory access. At block 1605, the data of the current memory access is loaded into the LRU entry of the CAM and the corresponding LM space from the external memory location identified by the base address. At block 1606, the data stored in the local memory is returned in response to the request. Figure 17 is a block diagram of an example of pseudo code for the process example 1600 of Figure 16. Other operations may also be performed.

[0063] Figure 18 is a block diagram of an example of a processor having multiple microengines according to one embodiment. The processor example 1800 includes multiple microengines 1801-1802. Each of the microengines 1801-1802 includes CAMs 1803-1804 respectively, which are managed by LRU logic 1807-1808 respectively. Each microengine further includes LMs 1805-1806 respectively, which are managed by index logic 1809-1810 respectively. The microengines 1801-1802 may be used to perform automatic software controlled caching operations described above, where each of the microengines may perform such operations for a respective thread substantially concurrently. It will be appreciated that some well-known components of a processor are not shown in order not to obscure embodiments of the invention in unnecessary detail.

[0064] Figure 19 is a block diagram illustrating an example of a data processing system according to one embodiment. The exemplary system 1900 may be used to perform the process examples for automatic software controlled caching described above. Note that while Figure 19 illustrates various components of a computer system, it is not intended to represent any particular architecture or manner of interconnecting the components, as such details are not germane to the present invention. It will also be appreciated that network computers, handheld computers, cell phones, and other data processing systems, which have fewer components or perhaps more components, may also be used with the present invention. The computer system of Figure 19 may, for example, be an Apple Macintosh computer or an IBM compatible PC.

[0065] Referring to Figure 19, the computer system 1900 includes, but not limited to, a processor 1902 that processes data signals. Processor 1902 may be an exemplary processor

100 illustrated in Figure 1. The processor 1902 may be a complex instruction set computer (CISC) microprocessor, a reduced instruction set computing (RISC) microprocessor, a very long instruction word (VLIW) microprocessor, a processor implementing a combination of instruction sets, or other processor device, such as a digital signal processor, for example. Figure 19 shows an example of an embodiment of the present invention implemented as a single processor system 1900. However, it is understood that embodiments of the present invention may alternatively be implemented as systems having multiple processors. Processor 1902 may be coupled to a processor bus 1910 that transmits data signals between processor 1902 and other components in the system 1900.

[0066] In one embodiment, processor 1902 includes, but not limited to, multiple microengines 1940-1942. The microengines 1940-1942 may be used to perform automatic software controlled caching for multiple threads substantially concurrently.

[0067] In addition, system 1900 includes a memory 1916. Memory 1916 may be a dynamic random access memory (DRAM) device, a static random access memory (SRAM) device, or other memory device. Memory 1916 may store instructions and/or data represented by data signals that may be executed by processor 1902. The instructions and/or data may include code for performing any and/or all of the techniques of the present invention. A compiler for compiling source code, including identifying one or more candidates suitable for software controlled caching, and inserting and expanding caching instructions to access the local memory rather than the external memory. Memory 1916 may also contain additional software and/or data not shown. A cache memory 1904 may reside inside or outside the processor 1902 that stores data signals stored in memory 1916. Cache memory 1904 in this embodiment speeds up memory accesses by the processor by taking advantage of its locality of access.

[0068] Further, a bridge/memory controller 1914 may be coupled to the processor bus 1910 and memory 1916. The bridge/memory controller 1914 directs data signals between processor 1902, memory 1916, and other components in the system 1900 and bridges the data signals between processor bus 1910, memory 1916, and a first input/output (I/O) bus 1920. In some embodiments, the bridge/memory controller provides a graphics port for

coupling to a graphics controller 1912. In this embodiment, graphics controller 1912 interfaces to a display device for displaying images rendered or otherwise processed by the graphics controller 1912 to a user. The display device may include a television set, a computer monitor, a flat panel display, or other suitable display devices.

[0069] First I/O bus 1920 may include a single bus or a combination of multiple buses. First I/O bus 1920 provides communication links between components in system 1900. A network controller 1922 may be coupled to the first I/O bus 1920. The network controller links system 1900 to a network that may include a plurality of processing system and supports communication among various systems. The network of processing systems may include a local area network (LAN), a wide area network (WAN), the Internet, or other network. A compiler for compiling source code can be transferred from one computer to another system through a network. Similarly, compiled code that includes the directives or instruction inserted by the compiler can be transferred from a host machine (e.g., a development machine) to a target machine (e.g., an execution machine).

[0070] In some embodiments, a display device controller 1924 may be coupled to the first I/O bus 1920. The display device controller 1924 allows coupling of a display device to system 1900 and acts as an interface between a display device and the system. The display device may comprise a television set, a computer monitor, a flat panel display, or other suitable display device. The display device receives data signals from processor 1902 through display device controller 1924 and displays information contained in the data signals to a user of system 1900.

[0071] A second I/O bus 1930 may comprise a single bus or a combination of multiple buses. The second I/O bus 1930 provides communication links between components in system 1900. A data storage device 1932 may be coupled to second I/O bus 1930. The data storage device 1932 may include a hard disk drive, a floppy disk drive, a CD-ROM device, a flash memory device, or other mass storage devices. Data storage device 1932 may include one or a plurality of the described data storage devices.

[0072] A user input interface 1934 may be coupled to the second I/O bus 1930, such as, for example, a keyboard or a pointing device (e.g., a mouse). The user input interface 1934

may include a keyboard controller or other keyboard interface device. The user input interface 1934 may include a dedicated device or may reside in another device such as a bus controller or other controller device. The user input interface 1934 allows coupling of a user input device (e.g., a keyboard, a mouse, joystick, or trackball, etc.) to system 1900 and transmits data signals from a user input device to system 1900.

[0073] One or more I/O controllers 1938 may be used to connect one or more I/O devices to the exemplary system 1900. For example, the I/O controller 1938 may include a USB (universal serial bus) adapter for controlling USB peripherals or alternatively, an IEEE 1394 (also referred to as Firewire) bus controller for controlling IEEE 1394 compatible devices.

[0074] Furthermore, the elements of system 1900 perform their conventional functions well-known in the art. In particular, data storage device 1932 may be used to provide long-term storage for the executable instructions and data structures for embodiments of methods of dynamic loop aggregation in accordance with embodiments of the present invention, whereas memory 1916 is used to store on a shorter term basis the executable instructions of embodiments of the methods of dynamic loop aggregation in accordance with embodiments of the present invention during execution by processor 1902.

[0075] Although the above example describes the distribution of computer code via a data storage device, program code may be distributed by way of other computer readable mediums. For instance, a computer program may be distributed through a computer readable medium such as a floppy disk, a CD ROM, a carrier wave, a network, or even a transmission over the Internet. Software code compilers often use optimizations during the code compilation process in an attempt to generate faster and better code.

[0076] Thus, automatic software controlled caching generations in network applications have been described herein. In the foregoing specification, the invention has been described with reference to specific exemplary embodiments thereof. It will be evident that various modifications may be made thereto without departing from the broader spirit and scope of the invention as set forth in the following claims. The specification and drawings are, accordingly, to be regarded in an illustrative sense rather than a restrictive sense.